SCRIPTING AND TEXT MANIPULATION (AND CONDOR!)

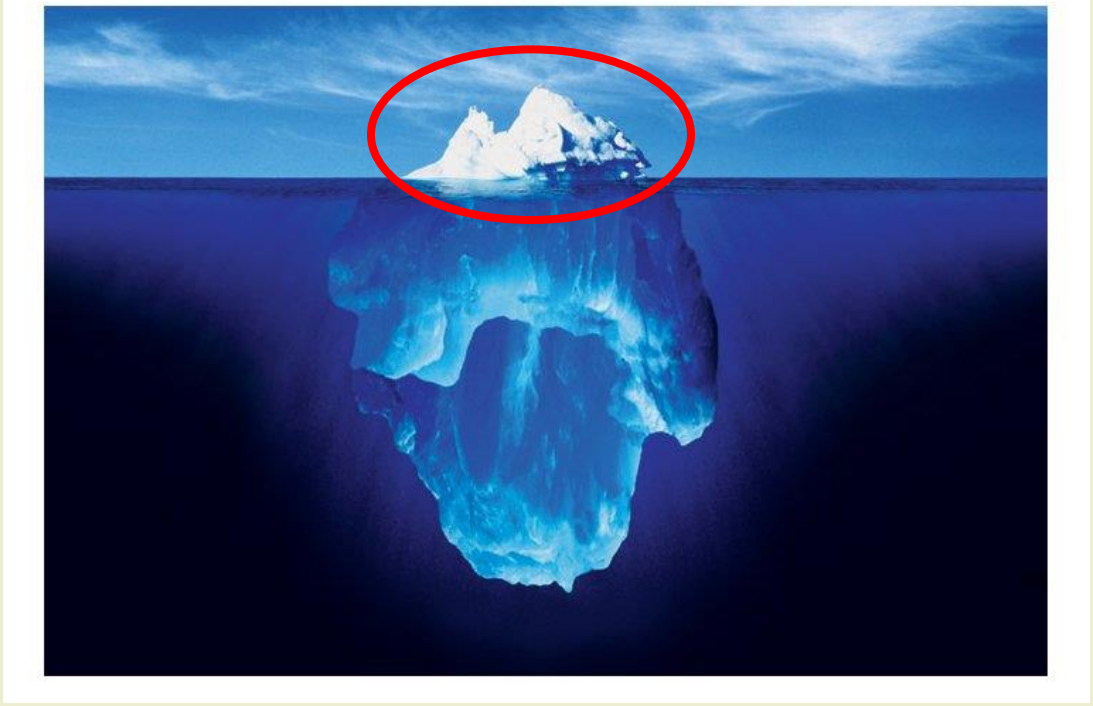
DANIEL JUMPER & MIKE BEAUMIER

Part 2 of 2:

Scripting and condor examples

SCOPE OF TALK

- Broad and shallow again!
  - Focused on providing varied examples for your reference



OVERVIEW

- Text Manipulation
  - Regular Expressions
  - sed
  - awk
- Scripts
  - Condor – batch program running on many computers
  - Bash – More simple, easy to use command line tools
  - Perl – Higher level; more structured programming

Part 1 Last Week

CONDOR

- Lets you run many instances of a program over many computers
- DISCLAIMER: I am giving you a limited scope of information about using condor. There are many other things you can do that I do not tell you about. Investigate for yourself and ask others!
- To submit jobs, you need:
  - .cmd file – program or command you want to execute
  - .job file – specify parameters to condor system to run jobs
  - `condor_submit` (or script!) – submit .job files to run on condor
- After you have submitted jobs:
  - `condor_status -submitters` Check the status of jobs per user
    - `alias mycondorstatus 'date; condor_status -submitters | grep danielj'`
  - `condor_q` more detailed information about individual jobs (flag options available)
  - `condor_rm danielj` remove jobs from the queue
  - `condor_release danielj` re-submit jobs that are held
- See scripting examples for more information on submitting jobs!

SCRIPTING: BASH AND PERL

- Two of many possible scripting languages
- Bash: quick and simple scripts
  - Simple and limited structure
  - Naturally accepts command line commands
- Perl: more potential for complex scripts or programs
  - Better higher level programming features
  - Can still do simple things, but is more like a full programming language

BASH BASICS

- Bash is a shell (terminal) environment
  - Syntax is similar to command line (most things you can do in a bash script you can do on the command line with ; separating commands)
    - NOTE: rcf uses csh instead of bash for the shell. They are very similar with minor differences in syntax
  - You can directly execute command line commands in bash
    - Simply type a command as you would in the command line. Each command on a new line
    - Alternately, use ` (the non-shifted ~ key) to execute a command within another line of the script
      - Eg: `variable=`ls | tail 1``
- Define what scripting language you're using in the first line
  - #!/bin/bash
- Whitespace doesn't matter much. Start new commands on new lines
  - Use \ to continue a single command over multiple lines
- Variables:
  - No types are defined
  - Define a variable with `variable=...` (no space before =)
  - Reference a variable with `$variable` or `${variable}`

BASH EXAMPLES

- 3 Examples:
  - hadd_merger: a script to merge *many* root files via hadd commands
    - Repository: offline/analysis/danielj/example_code/scripts/hadd_merger
  - simple_condor: a very basic example of a script submitting a few jobs to condor
    - Repository: offline/analysis/danielj/example_code/scripts/simple_condor
  - advanced_condor: a more complex script submitting many programs divided between a few condor jobs
    - Repository: offline/analysis/danielj/example_code/scripts/advanced_condor
- Accessing the repository:
  - https://www.phenix.bnl.gov/viewvc/viewvc.cgi/phenix/
  - Checking out a copy of the code:
    - cvs co –d <local directory name> <repository path to check out>
    - cvs co –d hadd_example offline/analysis/danielj/example_code/scripts/hadd_merger

BASH: HADD EXAMPLE

offline/analysis/danielj/example_code/scripts/hadd_merger

- Purpose: look for many root files in a directory (more files than the hadd command accepts) and use multiple hadd commands to combine them
- Highlighted techniques:
  - Input arguments
    - Pass arguments separated by spaces when executing the script:
      - EG: `$>  script.sh arg1 arg2 arg3`
    - `$1, $2, … , $n`
    - Set a default value for an argument if it's not present:
      - `${n:-<value>}` for example: `${1:-3}` or `${1:-"test string"}`
  - Do math in bash with `$(( <math here> ))` or `$[ <math here> ]`
    - eg. `$((1+2))` or `$[$var*3]`
  - Using ` ` ` to execute command line commands
    - Defining variables (basename command)
    - Looping over output of a command (find command)
  - One (of multiple) possible for loop syntax
  - Conditional (if) syntax
    - -gt greater than, –eq equals, -ge greater or equal, ……
- Disclaimer: make sure to change the hard coded scratch directory to your space!

BASH: SIMPLE_CONDOR EXAMPLE

offline/analysis/danielj/example_code/scripts/simple_condor

- Purpose: the most basic version of submitting jobs to condor
- Highlighted techniques:
  - Use *very* simple script makefilelist.sh to create a list of files
  - run_condor.sh is a simple script that, when executed, loops over files in a list and submits a job to condor for each one
  - Passing arguments to condor
    - Set parameters in condor.job OR with –a "…" flag for condor_submit in the script
      - "Executable" specifies what to execute (condor.cmd in this case)
      - "Initialdir" specifies from which directory to call execution
      - "Arguments" variable is what is passed to condor.cmd when executed
  - condor.job defines condor parameters
    - "Notification" and "Notify_User" can be used to email you status of the jobs
  - condor.cmd is what is executed for each job (in this case, it's a very simple bash script to read one file and output to another)

- DISCLAIMER: again, make sure you change paths (and my email address) that are coded in several of the files)

BASH: ADVANCED_CONDOR EXAMPLE

offline/analysis/danielj/example_code/scripts/advanced_condor

- Purpose: run on condor with many program executions split in batches between several condor jobs.
- Highlighted techniques:
  - fake_input_files/generate_files.sh – a simple script that generates text files
  - run_condor.sh dynamically generates a condor.cmd file to match how many executions you want to run per job
    - It decides how many arguments to pass to condor.cmd and modifies the text of condor.cmd to use the arguments and do the proper number of executions
    - When putting text in condor.cmd be, you have to be very careful about escaping characters with \
  - condor.cmd executes root macros with input and output files passed as arguments
    - Note proper syntax of escape characters
- DISCLAIMER: as usual, you have to modify paths and such if you want to check out the code and run it yourself

WHAT IS PERL

Perl slides from Mike Beaumier

- Perl is an interpreted, dynamic, high level programming language, well suited for quick and dirty scripts, as well as sophisticated software solutions
  - Interpreted: statements are executed line by line by an interpreter, instead of being parsed and compiled into a binary, executable machine-code.
  - Dynamic: data structures do not have a fixed size, and can be resized as needed. Code can crash when you run out of available memory
  - High Level: The user does not have to concern themselves with direct memory management (i.e. C/C++), programming syntax is heavily abstracted from what is actually executed by the processor.

THE ANATOMY OF A PERL SCRIPT

Perl is an extremely flexible and fluid scripting language. There are many ways to format a perl script. The convention presented here is not the "only" way to do it, but I think its less error prone.

Follow along with "Example 1" available from CVS:
cvs co offline/analysis/beaumim/tutorials/perl

A Perl script, from top-to-bottom, contains the following pieces:

- The "shebang" – First line of the script: #! /usr/bin/perl
- Extra libraries / Parsing Rules: use strict;
- Function prototypes
- Body
- Function Definitions

THE ANATOMY OF A PERL SCRIPT

```
 1 #! /usr/bin/perl
 2
 3 use strict; # this forces us to define variables before using them. Perl actually
 4 # allows us use undefined variables and will try to guess at what var
 5 # we are using if we don't explicity define it. Always use strict!
 6
 7 # local variables are defined with "my". Global variables are defined with "our".
 8 # Variables defined out of a scope: "{}" are global anyway.
 9
10 # Subroutine prototype. Usually, you'll want to document what the subroutine does.
11 # Subroutine - returns the average of a list of arguments. Arugments will be
12 # compressed into a single list - such that you can provide a mixture of scalars
13 # and arrays.
14 sub Average
```

Line 1 – The shebang – tells the interpreter where the perl executable lives
Line 3 – Additional rules – this one is “strict”
Line 4 – A Subroutine prototype or definition

The rest of the script constitutes the “body” and is simply just commands that are executed in order. One may organize the control flow using subroutines objects, or other scripts that are included after the shebang.

HOW IS PERL USED AT PHENIX

- General purpose scripting language
- Text parsing
- Online Data Production management
- Database interface and management
- Automation

Perl is often referred to as the “Swiss Army Chainsaw” of programming/scripting languages because of its versatility and wide support for a huge number of specific libraries and packages. The Perl philosophy might best be summarized as “There’s more than one way to skin a cat”.

BASIC PERL DATA STRUCTURES

Data structure "types" are indicated with sigils – a prefix that is either "$", "@", or "%". Commented lines begin with "#".

See Example 1

- Scalar – "$"
  - Can be a floating point, integer, string or interpreted string. In interpreted strings, certain special characters need to be "escaped" with a "\" character.
- Array – "@"
  - A collection of scalar variables. Each entry will be interpreted as a number or a string, depending on how it was defined or entered into the array.
- Hash – "%"
  - A data structure which pairs a lookup key value, with a mapped value
  - All about hashes: http://www.cs.mcgill.ca/~abatko/computers/programming/perl/howto/hash/

BASIC PERL DATA STRUCTURES

```
# Scalar Variables
my $var_1 = 123;
my $var_2 = 12.3;
my $var_3 = "abc";
my $var_4 = 'def';
my $var_5 = "1.23";
```

```
# Array Variable
my @array_variable = ($var_1, $var_2, $var_3, $var_4);
```

```
# Hash Variable
# There are many ways to create hashes. Hashes are a key-value mapping type of
# data structure. The value can be a scalar, or an array reference, or another
# hash. This data structure is very powerful, but also somewhat tricky.
# Hash keys must be unique.
my %hash_var = ( "var_1" => $var_1, # maps the key "var_1" to the value $var_2
  "var_2" => $var_2,
  $var_3 => "var_3", # variables themselves can be keys.
  "the_string_number" => $var_5
);
```

"ADVANCED" PERL

- References (not covered, similar to pointers in C++)
- Automatically Created Variables
  - "$_" and "@_"
  - $_ is a scalar variable or variable reference
  - @_ is an array
- Control Structures
  - Subroutine
    - A function. Can take a variable number of arguments, which are stored in the automatically defined variable "@_"
  - Loops
    - "for", "foreach"
    - "while", "until", "do..while"
    - Control statements: "next", "last", "continue", "redo", "goto"
    - http://www.tutorialspoint.com/perl/perl_loops.htm
  - Recursion (not covered)
  - "if", "else", "elsif" (covered in regex portion of talk). Not covered: "unless"
  - Objects (not covered)
- Objects (not covered)
- File I/O (see example 2)

"ADVANCED" PERL - SUBROUTINE

```
# Subroutine prototype. Usually, you'll want to document what the subroutine does.
# Subroutine - returns the average of a list of arguments. Arugments will be
# compressed into a single list - such that you can provide a mixture of scalars
# and arrays.
sub Average
{
  my $n = scalar(@_); # @_ is an array of all arguments passed. It is automatically
                      # generated.
  my $sum = 0;
  foreach (@_)
  {
    $sum += $_; # $_ is an automatically generated scalar variable which points to
                # an element, accessed one at a time, from the list @_.
  }
  my $average = $sum / $n;
  print "The average of the $n element list is $average\n";
  return $average;
}
```

"ADVANCED" PERL – CONTROL LOOPS

```
# Looping through an array with foreach, using automatically created variable $_
foreach( @array_variable )
{
  print $_, "\n";
}
```

```
# In comparison optations, @array_variable returns the size of the array
# indexing of arrays starts at "0"
# Looping through an array via array index
for( my $i = 0; $i < @array_variable; $i++)
{
  print $array_variable[$i], "\n";
}
```

"while", "until", "do..while" also exist, but aren't covered here.

Loop iterations may be modified with control statements: "next", "last", "continue", "redo", "goto"

REGEX EXAMPLE WITH PERL

- See folder called “example_2” in the CVS checkout package to follow along.
- In general – you will
  1. Read a file, or an array of strings, with the intention of matching, in whole (or part) pieces of the string to mine data
  2. While reading through a file, check if the regular expression is a match, with:
     if($my_string =~ /$regular_expression_string/m) { # do stuff }

*caveat – any occurrence of “/” must be escaped with a “\” due to the match syntax in Perl already reserving “/”.

SCENARIO 1: PRODUCTION JOBS ARE FAILING…

- You're in charge of producing physics data from raw data, and your jobs are failing. You want to know how many jobs have failed.
- Unfortunately, all log, error, output, shell scripts and other production related files are in the same directory, and it's a mess.
- You need to do the following:
  1. Sort out the contents of the log area
  2. Figure out how many files have failed production
  3. Do further analysis of logs for production files which have not failed.

  See offline/analysis/beaumim/tutorials/perl/example_2/parse_logs.pl

SCENARIO 1: PRODUCTION JOBS ARE FAILING

```
my $cwd = getcwd();

print "Globbing all filenames in $cwd/logs into an array\n";
my @log_files = glob("$cwd/logs/*");
my $dir_entries = scalar(@log_files);
print "Found $dir_entries logs\n";

# Now that we have our log files - we can sort them - with regular expressions!

my @logs = (); # sort .log files here
my @jobs = (); # sort .job files here
my @outs = (); # sort .out files here
my @errs = (); # sort .err files here
foreach ( @log_files )
{
  chomp $_; # remove any trailing newline characters
  if( $_ =~ /\/.*\/.*\.log/m ) # need to escape any / inside of the / /m match brackets
  {
    push @logs, $_;
  }
  elsif( $_ =~ /\/.*\/.*\.err/m )
  {
    push @errs, $_;
  }
  elsif( $_ =~ /\/.*\/.*\.out/m )
  {
    push @outs, $_;
  }
  elsif( $_ =~ /\/.*\/.*\.job/m )
  {
    push @jobs, $_;
  }
  else
  {
    print "$_ was not a file of interest.\n";
  }
}
```

Add all entries in a directory to an array

Use Regular Expression Matching to Filter Log Files By Type

SCENARIO 1: PRODUCTION JOBS ARE FAILING…

```
my @good_out_files = ();
my $bad_run_counter = 0;

foreach(@outs)
{
  open INFILE, "<", "$_" or die $!; # "<" opens a file for reading (line by line), ">" for writing, and ">>" for appending
  my $filename = $_;
  my $file_good = 1;
  while(<INFILE>) { # this usage reads through a text file line by line.
    chomp $_;
    if( $_ =~ /cannot find \/common\/[abc][0-9]\/eventdata\/EVENTDATA_P00-0000(\d{6})-(\d{4})\.PRDFF, exiting/m)
    {
      # Here, we introduce matching variables from regex. For each () in the expression,
      # a substring will be matched. The first, from left to right, is numbered $1,
      # the next, $2, and so on.
      print "    Run $1, Segment $2 was not found on disk\n";
      $bad_run_counter++;
      $file_good = 0;
    }
  }
  close INFILE;
  if( $file_good == 1 )
  {
    push @good_out_files, $filename;
  }
}
```

Here, we open each file, and read through it, testing each line for possible errors in reading a PRDFF. We can also sort log files by “good” or “bad”.

SCENARIO 2: THERE MAY BE BOTTLENECKS

- This was a real problem during the Run 12 production – many production jobs were failing, because of bottlenecks in the production chain.
- Producing physics data had a few steps: queuing PRDFFs in cluster, copying PRDFFs to cluster, and running production macro over PRDFFs.
- No information was recorded about this process, other than some time-stamps buried in tens of millions of lines of log-file output.
- We needed to determine where bottlenecks in the chain were, by extracting these time stamps from log files, and studying the time distributions for each step.

See offline/analysis/beaumim/tutorials/perl/example_2/parse_logs.pl

SCENARIO 2: THERE MAY BE BOTTLENECKS

```
my @data_set = (); # This data set will contain: run_number seg_number queue_time copy_time prod_time
foreach(@good_out_files) {
  # We need to extract the run number and segment number from our file-name, so that
  # we can use this to correlate with our time-stamps.
  my $run = 0;
  my $seg = 0;
  if( $_ =~ /\/.*\/EVENTDATA_P00-0000(\d{6})-(\d{4}).out/m )
  {
    print "$1-$2:\n";
    $run = $1;
    $seg = $2;
  }
  my @time_stamps = (); # blank out time stamps
  open INFILE, "<", "$_" or die $!;

  while(<INFILE>) {
    chomp $_;
    if( $_ =~ /\S{3}\s\S{3}\s+\d+\s+\d+:\d+:\d+\s*(EDT|EST)\s*\d{4}/m)
    {
      # ParseDate converts format to: YYYYMMDDHH:MM:SS
      # UnixDate returns format according to flags in argument, here its just seconds (unixtime)
      my $unixtime = UnixDate(ParseDate($_),"%s");
      push(@time_stamps,$unixtime);
      print "$_: $unixtime \n";
    }
  }
  close INFILE;
  my $queue = $time_stamps[1] - $time_stamps[0];
  my $copy  = $time_stamps[2] - $time_stamps[1];
  my $prod  = $time_stamps[3] - $time_stamps[2];
  my $results = "$run\t$seg\t$queue\t$copy\t$prod";
  push @data_set, $results;
}
```

Extract Run Number and Segment Number From Filename

Extract and convert time stamps to seconds

Use extracted data for useful calculations!

ANOTHERUSAGE EXAMPLE OF PERL IN ANALYSIS

- Mike Beaumier developed a specific Perl tool to automate the creation of relevant bash scripts, condor job files, and environment configuration for cases where a single root script is called with a large variety of arguments.
- Check out a copy from CVS to play with (with working example, README, etc) here:
  - offline/analysis/beaumim/condor
- And send suggestions / bugs / questions to:
  - Mike Beaumier – michael.beaumier@gmail.com

END